

FAQ

Q What is JavaScript and how does it work?

A JavaScript is a scripting language that can be used in the development of webpages. It's most often used as a client-side scripting language, which means that the web browser, not the server, runs the code.

JavaScript is interpreted, so it doesn't need to be compiled. One bit of code can be run on any computer with a compatible web browser regardless of operating system. This makes JavaScript incredibly powerful.

It's often used to add extra features to websites, such as dynamic menus that extend when clicked on. This gives web applications more of a desktop feel. The technology is free to use and built into web browsers, so you can develop JavaScript using Notepad. In this month's *Programming Expert*, we'll demonstrate its capabilities.

Mike James

IT author, lecturer
and programmer

mike@computershopper.co.uk



Building JavaScript controls

HTML can get complex and out of hand. Mike James shows you how to use JavaScript to simplify webpage creation



There are so many technologies available when building webpages, it's difficult to know what to use. Most approaches seem complex and you have to spend a long time learning new languages, techniques and configuration options. One of the simplest approaches to building active webpages (webpages that react and change based on user input) is to use client code in JavaScript called Spartan Ajax.

This approach is not well known, but you can visit <http://spartanajax.iopress.co.uk> for details. It is simple as long as you know how to program in JavaScript and are prepared to explore the Document Object Model (DOM). Even if you decide to use another technological framework to produce your webpages, the ideas in Spartan Ajax are still worth knowing, because fragments of the JavaScript code can be used elsewhere.

The key idea in Spartan Ajax is to avoid using HTML and do as much as possible in code. This simplifies what you are doing, as there is no artificial split into markup and procedural language. All the code that makes the page work is in one place, and you can treat the whole thing like a desktop application. All the technology is standard, free and readily available. Using just JavaScript to create webpages reduces the task to procedural programming, without having to buy anything new.

Spartan Ajax's motto is 'No more HTML', but this is a little extreme. HTML is great if you want to produce a page with lots of text, pictures and links. However, if you are trying to produce a page that is more like a desktop application, HTML forces you to manage the multiple technologies and make them work together. We also have the problem of browser compatibility, a universal problem that affects HTML as well as JavaScript and the DOM. The examples listed here have been tested on Internet Explorer 7 and Firefox 2, which accounts for around 97 per cent of the market.

GETTING STARTED

All you need is a text editor such as Notepad, but you can use an HTML/JavaScript editor such as Visual Web Developer 2008 Express Edition, which can be

downloaded for free from www.microsoft.com/express/vwd. We could do away with HTML altogether, but even a Spartan Ajax application benefits from having a standard HTML page that loads the code and presents something that non-JavaScript browsers can load:

```
<html>
<head>
  <script type="text/javascript"
    src="Sparta.js">
  </script>
  <script type="text/javascript"
    src="demo1.js">
  </script>
  <title>Spartan Testbed</title>
</head>
<body>
</body>
</html>
```

This is the only HTML page we need. You should create it and save it as demo.html or demo.htm in a suitable folder. The page loads two scripts: sparta.js, which contains the standard object code that we are going to use, and demo1.js, which uses it.

MAKING JAVASCRIPT WRAPPERS

We are going to be creating JavaScript wrappers for DOM objects. To do this easily and efficiently, we are going to be using object-oriented JavaScript and a few clever tricks. All the familiar DOM objects such as buttons, tables, selection lists and so on have corresponding JavaScript objects, and these are used to create the wrappers.

Let's take a simple example and wrap a button:

```
button=function(parent)
{
  var DOMObj=document.createElement
    ("button");
  DOMObj.style.position="absolute";
```

```
parent.appendChild(DOMObj);  
return DOMObj;  
}
```

This is almost a standard JavaScript object constructor that creates a new DOM button, sets its positioning to absolute and then appends it to a parent object already in the DOM hierarchy. However, this constructor returns DOMObj and not the default 'this' reference. This makes the new JavaScript object an instance of the JavaScript class that represents the DOM object.

By the same reasoning, any variables or methods that you create must be part of the DOM object, so they too must use the DOMObj reference rather than this. This new button class is entered into the sparta.js file and in the demo.js file we can use it with code like this:

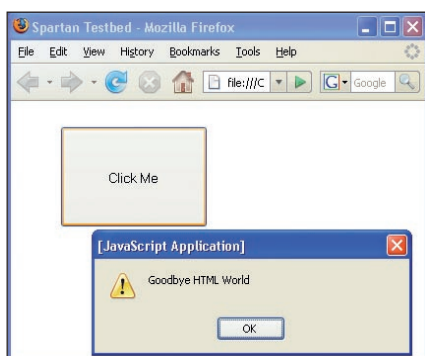
```
button1=new button(document.body);  
button1.style.top=25;  
button1.style.left=50;  
button1.style.width=150;  
button1.style.height=100;  
button1.innerHTML="Click Me";  
button1.onclick=function()  
{  
    window.alert("Goodbye HTML World");  
}
```

If you load the HTML page you will see a button, and clicking it produces the alert box. It's very similar to the way you would create a desktop application using Java, VB6 or any .NET language. Everything is visible, there is no need to switch from HTML to JavaScript, all the names are visible, there is no code lurking in obscure bits of HTML and no HTML lurking in the code.

THE FRAMEWORK

There are lots of ways to wrap the DOM objects. You can change the Sparta framework to suit your programming style. The one presented has the advantage of being simple and requiring little code. As all the basic DOM input objects can be created in the same way, it makes sense to factor out the code that does the creation:

```
Sparta=new Object;  
Sparta.DOMInputObject=function  
(parent,type)  
{  
    var DOMObj=document.  
        createElement("input");  
    DOMObj.type=type;  
    DOMObj.style.position="absolute";  
    parent.appendChild(DOMObj);  
    return DOMObj;  
}
```



▲ The new button class

We create a Sparta object to make a namespace that can be used for 'private' internal classes and methods used by the framework itself. In this case, we have defined a general object creation method. The button class can then be written:

```
button=function(parent)  
{  
    var DOMObj=Sparta.DOMInputObject  
        (parent,"button");  
    return DOMObj;  
}
```

All the simple DOM input classes are created in the same way. For example, a text input field is just:

```
textbox=function(parent)  
{  
    var DOMObj=Sparta.DOMInputObject  
        (parent,"text");  
    return DOMObj;  
}
```

A password input box is:

```
passwordbox=function(parent)  
{  
    var DOMObj=Sparta.DOMInputObject  
        (parent,"password");  
    return DOMObj;  
}
```

For general DOM objects – controls that are not based on the input object – we can use something similar:

```
Sparta.DOMObject=function(parent,  
    name)  
{  
    var DOMObj=document.createElement  
        (name);  
    DOMObj.style.position="absolute";  
    parent.appendChild(DOMObj);  
    return DOMObj;  
}
```

To create a text area such as a multi-line text box, we would use:

```
textarea=function(parent)  
{  
    var DOMObj=Sparta.DOMObject(parent,  
        "textarea");  
    return DOMObj;  
}
```

Once you have seen these two in action, it's clear that they can be rewritten as a single routine that serves to create any DOM object:

```
Sparta.DOMObject=function(parent,  
    name,type)  
{  
    var DOMObj=document.createElement  
        (name);  
    if(type)  
    {  
        DOMObj.type=type;  
    }  
    DOMObj.style.position="absolute";  
    parent.appendChild(DOMObj);  
    return DOMObj;  
}
```

Once you start working in this way, you should see that there are many simplifications you can make to the controls available. After all, what is the difference between a text input and a text area? To most programmers, they seem to do the same job. However, a text area is multi-line and text box single line.

Many would think it's easier to add a single multi-line property to the text box class and use it to control whether it's a text input or area:

```
textbox=function(parent,multiline)  
{  
    if(multiline)  
    {  
        var DOMObj=Sparta.DOMObject  
            (parent,"textarea");  
        DOMObj.multiline=true;  
    }  
    else  
    {  
        var DOMObj=Sparta.DOMObject  
            (parent,"input","text");  
        DOMObj.multiline=false;  
    }  
    return DOMObj;  
}
```

The multi-line property is read-only, in the sense that you can use it to test if the control is multi-line or not, but changing its value doesn't change the behaviour of the control. You can make almost any property read/write, but you need extra code to change the control to reflect the current setting.

COMPOUND CONTROLS

Some controls are built up using multiple HTML tags. This is something Spartan Ajax can simplify. We can opt to bundle the various objects together in the wrapper, so the whole control is created in one step. To create a drop-down list in HTML, you have to use a select tag that contains a number of options tags – one for each option in the list. We can wrap the selection control, so all the programmer has to do to create it is provide a list of options as an array:

```
select=function(parent,list)  
{  
    var DOMObj=Sparta.DOMObject  
        (parent,"select");
```

Once we have created the select object, we can loop through each of the options specified in the array, create an option object for each one and add it to the select object:

```
for(var i=0;i<list.length;i++)  
{  
    var op=document.createElement  
        ("option");  
    op.value=i;  
    op.text=list[i];  
    try  
    {  
        DOMObj.add(op,null);  
    }  
    catch(ex)  
    {  
        DOMObj.add(op);  
    }  
    return DOMObj;  
}
```

The only complication is that IE doesn't implement the add method correctly, so we need a try-catch construct to see which form of 'add' will actually work.

With this added to the framework, we can create a selection control very simply:

```
select1=new select(document.body,
    new Array("Option one",
        "Option two"));
select1.style.top=25;
select1.style.left=50;
```

Creating lists with any number of options is just as easy.

THE TABLE OBJECT

We could easily continue in this way to construct composite controls one by one until all the standard HTML is available as JavaScript classes. However, it is actually far more interesting to tackle one of the bigger tasks: to create a table class and demonstrate how custom controls can be built by creating a calendar control.

The table object is created in HTML using a table tag that contains lots of <tr> tags (one pair for each row) and lots of <td> tags (one pair for each cell). The same task can be done easily in JavaScript by using a constructor that specifies the number of rows and columns the table should have.

Creating the table object is straightforward:

```
table=function(parent,r,c)
{
    var DOMObj=Sparta.DOMObject
    (parent,"table");
    DOMObj.border=1;
```

Then all we have to do is create r row objects and add c cell objects to each row. This is just a matter of two For loops. The first creates the row objects:

```
for(var i=0;i<r;i++)
{
    var row=DOMObj.insertRow(i);
```

The second adds the cell objects to each of the rows:

```
for(var j=0;j<c;j++)
{
    var col=row.insertCell(0);
    col.innerHTML="&nbsp;";
}
return DOMObj;
```



▲ The easy way to create a select list

With this new definition, if we want to create four rows and seven columns we just use:

```
table1=new table(document.body,4,7);
table1.style.top=25;
table1.style.left=50;
```

We can also access any cell in the table to use its content with an instruction such as:

```
table1.rows[2].cells[3].
    innerHTML="Table test";
```

It is also easy to add a method to the table class that returns or sets the contents of the cell:

```
DOMObj.cellvalue=function(r,c,text)
{
    if(text)
    {
        this.rows[r].cells[c].
            innerHTML=text;
    }
    else
    {
        return this.rows[r].cells[c].
            innerHTML;
    }
    return DOMObj;
}
```

You can then set the content of a cell using:

```
table1.cellvalue (3,2,"More text");
```

You can return the content of a cell using:

```
alert(table1.cellvalue (2,3));
```

You can even write:

```
table1.cellvalue (1,1,table1.
    cellvalue (2,3));
```

This copies the contents of row 2, column 3 to row 1, column 1.

You can continue to add methods and properties to make using a table easier. One method that is indispensable, however, is to return a pointer to a cell object at a given row and column:

```
DOMObj.cell=function(r,c)
{
    return this.rows[r].cells[c];
}
```

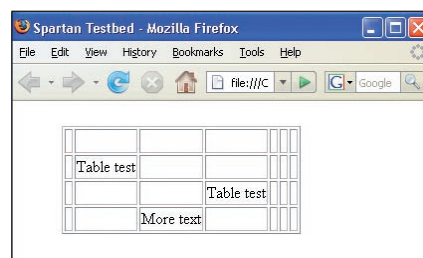
Once you have the cell reference you can do anything you want to the cell by getting directly at its properties and methods.

MAKING A CUSTOM CALENDAR

Our final example demonstrates one of the real powers of JavaScript building controls. Not only can you wrap existing HTML controls, but you can also extend what is available by creating custom controls. HTML has no calendar tag so there is no calendar class within the DOM, but it is possible to create one based on the table class that we have just developed.

The first thing we need is the table:

```
calendar=function(parent,month,year)
{
    var DOMObj=new table(parent,6,7);
```



▲ An easy-to-use table

```
var DOMObj=new table(parent,6,7);
```

The top row is going to contain the days of the week, and setting the width of its cells sets the width for each column:

```
var width="30px";
for(var i=0;i<7;i++)
{
    DOMObj.cell(0,i).width=width;
}
```

The most direct way to populate these cells is:

```
DOMObj.cellvalue(0,0,"Sun");
DOMObj.cellvalue(0,1,"Mon");
DOMObj.cellvalue(0,2,"Tue");
DOMObj.cellvalue(0,3,"Wed");
DOMObj.cellvalue(0,4,"Thr");
DOMObj.cellvalue(0,5,"Fri");
DOMObj.cellvalue(0,6,"Sat");
```

At this point, we have a table with fixed width columns displaying the days of the week in the top row.

Our next problem is standard and has little to do with JavaScript or using the DOM. We need to work out which day the first day of the month will be. This is just an exercise in using date and time functions – specifically, JavaScript's date object and its associated methods.

First, we need a date object corresponding to the first day of the month for which we are constructing the calendar:

```
var d=new Date(year,month-1,1);
```

JavaScript has January as month zero. This date object is used to get the weekday number, counting Sunday as day zero:

```
var FirstDay=d.getDay();
```

The weekday number gives us the column in which the calendar starts. We now need a For loop to step through the number of days in a month. There isn't a simple way of discovering the number of days in a month. We know that there can't be more than 31, however, so the best solution is to create a For loop that always steps through 31 days, and simply ignore the loops where the month changes:

```
for(var day=1;day<32;day++)
{
    d.setDate(d.getDate()+1);
    if(d.getMonth()+1==month)
    {
        day=1;
    }
}
```

The setDate method sets the date object to the day specified, but if the day goes beyond the number of days in the month then the month rolls over to the first day of the next month. So

SAMS

Microsoft XNA Unleashed

★★★★★

£29

From www.amazon.co.uk

Microsoft's XNA is a set of tools for developing computer games for the Xbox 360 and Windows. *Microsoft XNA Unleashed* covers how to use the tools and introduces games programming.

Everything covered in this book (including C# Express, DirectX 9 runtime and XNA Game Studio Express) can be downloaded for free, and there is advice on installing them all. The book explains how to use the tools just to develop for Windows PCs. If you're interested in console development, though, there's plenty on this, plus information on how to share code between the two platforms for shorter development times. This is a highly practical book that progresses logically, covering XNA basics for creating 3D objects, then input devices and camera, before moving on to the Content Pipeline, sounds and music. It shows how to create, compile, debug and deploy games in 2D and 3D, and covers topics such as physics, artificial intelligence and special effects.

It explains how to distinguish between a work in progress and a playable 3D game, and all the C# examples and source code are included on a CD. If you're interested in developing games, it's a great place to start.



SUMMARY

- ☒ Well-paced guidance
- ☒ Not for the total beginner

ISBN 0672329646
PAGES 524

O'REILLY

Head First C#

★★★★★

£20

From www.amazon.co.uk

O'Reilly's Head Start series is a cross between a notebook and a joke book. The mix of conversational style, annotations and quiz elements seems to suit C# better than other topics. However, you must be prepared to work through the book, as the activities are not optional.

Beginners should find the book's style motivational. It's a great time to try C# 3.0, as it's available in the free Visual Studio 2008 Express Edition. This book shows how Visual Studio makes it easy to get started, and only tackles object-oriented concepts later on. The three hands-on labs in the book are games and, if you are pleased with your efforts, you can show them off on the book's website, from where you can also download the games' executables if you need extra help and support.

Head First C# covers the newest C# feature for querying data in .NET collections, LINQ, and puts it to work in the final lab (Space Invaders) to make collision detection easier. This project also makes good use of timers, another of the book's more advanced topics. On the whole, however, this isn't an advanced treatment. Even when it does tackle sophisticated ideas, it treats them in an informal manner. It's fair to say you'll probably need at least one other book about C#, though – one with a more staid approach.

SUMMARY

- ☒ Interactive for the beginner
- ☒ Very long

ISBN 0596514824
PAGES 778

as long as the current month agrees with the specified month, we can proceed to fill the table. First, we need to work out the row and column of the cell that will hold the date. We can easily work out the number of the cell that will hold the date by ignoring rows and columns, and treating the table as just one long string of cells:

```
var offset=FirstDay+day-1;
```

We can then convert this into the row and column numbers using simple arithmetic operators and functions:

```
var r=Math.floor(offset/7)+1;
var c=offset%7;
```

To see how this works, try evaluating it with a few test values of offset. Once we have the row and column, we can store the day of the month number and centre the entry:

```
DOMObj.cellvalue(r,c,day);
DOMObj.cell(r,c).align="center";
```

These two actions are repeated on each cell of the table that corresponds to a valid date in the same month.

ADDING CONTROL BEHAVIOUR

At this point, we could finish the For loop and try the calendar out. It would work, but it wouldn't be much use apart from allowing a user to see the dates in a month. To be more useful, the control must allow the user to select a date. We need to add behaviour to the control using event handling. Unfortunately, event handling is a non-standard and messy technology, and very few web browsers do it in the same way.

We need to respond to a user clicking on a cell by changing the appearance of the existing selected cell back to normal and the new

selected cell to bold – or reverse foreground background colours, or whatever you want to do to indicate which cell is selected. This means we have to associate an event handler with each cell. In theory, you can just write a single-click event handler for the table and discover which cell triggered the event. Browsers all handle this in different ways. Still, as there are only a maximum of 31 active cells, and what we are doing in response to each event is simple, defining 31 event handlers isn't so bad. Setting the onclick event handler for each cell is easy:

```
DOMObj.cell(r,c).onclick=
function(e)
{
    DOMObj.selected.style.
        fontWeight='normal';
    DOMObj.selected=this;
    DOMObj.selected.style.
        fontWeight='bold';
    DOMObj.selectedDate=selected.
        innerHTML;
    DOMObj.onDateChange();
}
}
```

The key to understanding how this event handler works is to know that when an event occurs on an object, the 'this' variable is set to the identity of the object. It is assumed that there is already a selected cell, and we set the font weight of the selected cell back to normal. By storing 'this', we create the newly selected cell and indicate this to the user by setting its font weight to bold. So we can see what date the user has selected, the day of the month is stored in a selectedDate property. The final line calls the onDateChange member function, which acts like a new event that we call when the selected date is changed. This also has to be initialised to a null

function to allow everything to work, even if an event handling function isn't supplied.

This ends the If statement and the For loop, but we have one more thing to do. There always has to be a selected date, so we need to add lines to select the first day automatically:

```
DOMObj.selected=DOMObj.cell
(1,FirstDay);
DOMObj.selected.style.
    fontWeight='bold';
DOMObj.selected.align="center";
DOMObj.selectedDate=1;
```

We also need to initialise the new event:

```
DOMObj.onDateChange=function(){};
```

Now all that remains is to return the pointer to the newly created calendar object:

```
return DOMObj;
```

We can then try it out:

```
call=new calendar(document.
body,12,2008);
call.style.top=100;
call.style.left=10;
call.onDateChange=function()
{
    alert(call.selectedDate);
}
```

This creates a calendar object and displays the day that the user has selected in an alert box.

This is still an unfinished control. It needs a method to set the month displayed without having to create a new calendar, but this is easy to do and is included in the code on this month's cover disc. **CS**